# 循环冗余检验（CRC）原理与实现
# （版本 1.1）

2005.9

# 申明

为了方便设计和使用EDB430及430混合信号处理器，编写了这个应用笔记。由于水平有限，难免有错漏之处，希望读者能够指点，以期不断完善。如果你要使用其中的文字，请注明来处，同时，本文作者不承担因用户在使用过程中造成各种错误的损失，也不提供其他任何承诺，这个文档仅供参考，不做为商业用途。

作者

2005-9-12

# 目录

# 1 前言

## 1.1 背景

循环冗余检验（Cyclic Redundant Check）已被广泛用于通信应用中，作为数据传输中差错控制的基本方法之一，各种书刊和杂志有很多介绍。本文档出于实用的目的，主要介绍其实现的原理，减少了原理的阐述，从而快速掌握和应用。以下是在 EDB430 实验开发平台上 CRC16 实现的过程。

## 1.2 CRC 的分类

目前 CRC 的使用分为标准和非标准两种，非标准为用户自定义 CRC 的生成多项式，而标准是已被国际标准化组织规定的标准生成多项式，这也是目前广泛使用的几种。

### 1.2.1 标准的 CRC

在通信协议中常见并被广泛使用的标准列于表中。(本文中以 16 位的 CRC-16 为例，除非另外说明)。

| 名称 | 多项式 | 简记 | 应用 |
|---|---|---|---|
| CRC-4 | $x^4+x+1$ | 0x13 | ITU G.704 |
| CRC-16 | $x^{16}+x^{15}+x^2+1$ | 0x8005 | IBM SDLC |
| CRC-CCITT | $x^{16}+x^{12}+x^5+1$ | 0x1201 | ISO HDLC, ITU X.25, SDLC, V.34/V.41/V.42, PPP-FCS |
| CRC-32 | $x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$ | 0x104C11DB7 | ZIP, RAR, IEEE 802 LAN/FDDI, IEEE 1394, PPP-FCS |
| CRC-32C | $x^{32}+x^{28}+x^{27}+x^{26}+x^{25}+x^{23}+x^{22}+x^{20}+x^{19}+x^{18}+x^{14}+x^{13}++x^{11}+x^{10}+x^9+x^8+x^4+1$ | 0x11EDC6F41 | SCTP |

表1-1 标准CRC多项式

### 1.2.2 非标准的 CRC

非标准的 CRC 一般是为了某种用途而采用不同于标准的生成多项式，而实际的操作原理是相同的，主要用于需要 CRC 而需要低成本的应用，或者为了减轻计算机处理负担而又能够保证数据可靠性的折中办法，此外，部分的加密算法也是用 CRC 来生成。

## 1.3 CRC 的原理简介

### 1.3.1 CRC 生成基本理论

实现 CRC 的基本原理，简单地说，就是原始数据通过某种算法，得到一个新的数据，而这个新的数据与原数据有着固有的内在关系。通过把原数据和新数据组

合在一起，形成新的数据，因此这个新数据具有自我校验的能力。我们把原来的数据表示为 P(x), 它是一个 n 阶的多项式，表示为：

$$P(x)= a_{n-1}x^{n-1}+a_{n-2}x^{x-2}+\cdots+a_1x+a_0$$

式中 $a_i$ 为 0 或 1，x 为伪变量，并用 $x^i$ 指明各位间的排列位置。因此，一个 8 位的二进制数 01001001 可以表示为:

$$P(x)=0x^7+1x^6+0x^5+0x^4+1x^3+0x^2+0x^1+1x^0$$

P(x)通过除以 CRC 多项式 G(x)（表 1-1 所示）后，得到一个余数 R(x)和商 Q(x), 这个 R(x)就是我们需要的 CRC 校验值，上述用公式表示：

$$\mathbf{P(x)=Q(x)*\ G(x)+R(x)} \qquad (1)$$

然而，上述公式仅适用于理论模型，而实际应用中，还需要考虑几个问题

首先要考虑 CRC 的数据位，不管 P(x)有多长的数据位，总是希望有一个固定位数的 R(x)，以便在实现时可以节省很多资源。同时考虑到 P(x)的数据位长度可能比 CRC 位数短，为了得到一个 16 位（或 32 位）的数据长度，必须将原有数据扩展到 16 位以上(表示为 $P(x)*X^r$)，才能到一个 16 位的余数。通常的做法是在 P(x)的右边添加相应的 CRC 位数，例如，16 位则需增加 16 个数据位，32 位需要增加 32 个数据位。

其次，规定 CRC 的最高位和最低为必须为 1，由于标准的 CRC 是 17 位和 32 位，因此，如果在目前 8 位，16 位，32 位，64 位等数据总线的计算机上实现非常时需要考虑。在实际使用中，我们并不需要考虑这个最高的 CRC 位，因为它是总是被舍去的，故只要考虑余下的 16 位（或 32 位）就可以了。

第三，商不需要，因此，根本不要考虑。

第四，除法运算没有数学上的含义，而是计算机中的模 2 算法，即，每个数据位，与除数作逻辑异或运算，因此，不存在进位或借位问题。

### 1.3.2 CRC 产生的操作过程

以下是一个 8 位的数据 0x02 产生一个 16 位的 CRC 的过程。

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | |
| | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | |
| | | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | |
| | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | |
| | | | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | |
| | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | |
| | | | | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| | | | | | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | | | | | | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | | | | | | | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | |
| | | | | | | | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| | | | | | | | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| | | | | | | | | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| | | | | | | | | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| | | | | | | | | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| | | | | | | | | | 8 | | | | 0 | | | | 0 | | | | F | | |

表 1-2，8 位数据 0x02 的 16 位 CRC 数据产生

在图中我们可以看到，这个 0x02 数据被扩充到 24 位（原数据 8 位+16 位 CRC，不足的用零填充），然后再与 0x8005（CRC-16 生成多项式）做模 2 运算。在运算过程中，第 17 位总是被舍去（图中红色的位）。第 16 位如果是零，那么，只与 0x0000 作异或运算，即数据左移一位。如果为一，那么，就要与 0x8005 做模 2（异或）运算。每次运算完毕，丢弃最高位，然后，将数据下一位移入，再进行新的模 2（异或）运算，直到所有的位移完为止。

### 1.3.3 CRC 检验基本理论

在接收端，接收的数据为 $P(x)*X^r+R(x)$，因此公式（1）表示为（2），由于 R(x)+R(x)的异或运算为零，则公式（2）可表示为（3）。如果将公式（3）除以 G(x)，则结果为：

$$\mathbf{P(x)}*X^r+\mathbf{R(x)=Q(x)*\ G(x)+R(x)+\ R(x)} \quad (2)$$

$$\mathbf{P(x)}*X^r+\mathbf{R(x)=Q(x)*\ G(x)} \quad (3)$$

$$\frac{P(x)*X^r+R(x)}{G(x)}=\frac{Q(x)*\ G(x)}{G(x)}=Q(x)$$

说明可以被 G(x)整除(最后异或运算结果为零)，因此，只要在接收端，对所有接收到的数据（包括 CRC 校验数据）除以 G(x)，如果余数为零，则说明数据传输无误。以刚才 0x02 数据（CRC=0x800F）为例，计算如下：

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | |
| | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | |
| | | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | |
| | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | |
| | | | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | | | |
| | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | |
| | | | | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | | |
| | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| | | | | | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | |
| | | | | | | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | |
| | | | | | | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| | | | | | | | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| | | | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | | | | | | | | 0 | | | | 0 | | | | 0 | | | | 0 | | | |

表 1-2，8 位数据 0x02 的 16 位 CRC 数据校验

当然，也可以使用比较直观的办法，只对与数据 0x02 求 CRC 值，如果等于接收到的 CRC，那么也说明数据没有问题，但这是个低效率的做法，不推荐使用。

# 2 实现

根据应用的需要，有两种常用的方法，来实现 CRC 的算法，逐位运算法和查表法，它们的特点如下。

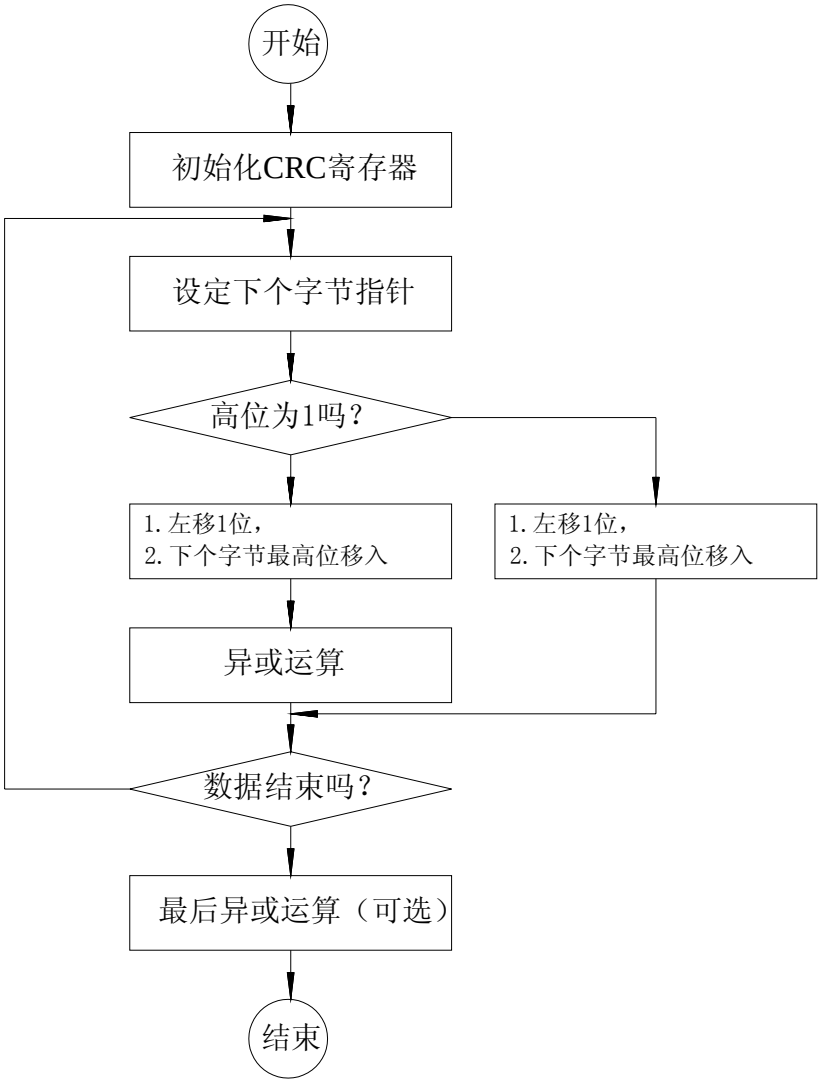
## 2.1 逐位运算法

逐位运算法，直接根据表 1-3 的原理实现的。数据每次移入一位时，就需要重新对每一位进行再次运算，因此：

移位操作的次数=总的数据位数-16

可以看出，当数据位数比较多时，CPU 使用的时间还是比较多的，数据的位数越多，就意味着使用更多的 CPU 机时，由此会带来了功耗和速度问题。

◆ 其 8 位数据信息实现 CRC 的流程图如下。




◆ 演示代码如下：

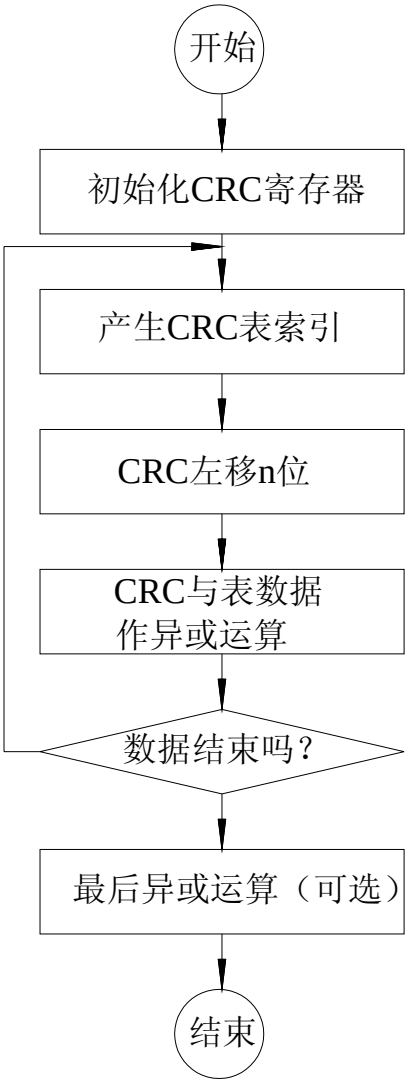
```
void CRC16_Bitwise(unsigned char *pMsg,unsigned char len)
```

```
{
unsigned char i,j;
 ......
   for(j=0;j<len;j++)
   {
    pMsg++;
    for(i=0;i<8;i++,*pMsg<<=1)
    {
       if(CRC16Temp & 0x8000)
       {
        CRC16Temp<<=1;
        CRC16Temp |=(*pMsg & 0x80)>>7;
        CRC16Temp ^=CRCPOLY16;
       }
       else
       {
        CRC16Temp<<=1;
        CRC16Temp |=(*pMsg & 0x80)>>7;
       }
    }
   }
......
}
```

## 2.2 查表法

查表法，是使用预先算好的基本 CRC 值，直接查出 CRC 值，因此，它是基于字节或字操作，避免了耗时的位运算。这就决定了它的速度会增加，由此带来功耗降低的好处，可这是以付出存储器为代价的，因为，必须预先在程序中存在一个 CRC 数据表。以 8 位数据为例，每一个字节仅需要作一次异或操作。表中的 CRC 值与其索引值有一个一一对应的关系。不像逐位法那样，每次移入一个位，就进行运算，查表法是每次移入一个字节，得到其索引值，然后，与这个缩影值做异或运算。粗略的看起来，所用的时间为逐位法的 1/5（具体根据计算机的指令周期而定）左右。

◆ 以下是八位数据 CRC 查表程序流程图。




◆ 演示代码如下：

```
void CRC16_Table(unsigned char *pMsg,unsigned char len)
{
unsigned char j;
unsigned int CRCindex;
......
for(j=0;j<len;j++,pMsg++)
   {
    CRCindex=(CRC16Temp>>8) ^ *pMsg;
    CRC16Temp<<=8;
     CRC16Temp ^=CRC16Table[CRCindex];
   }
......
}
```

### 2.3 CRC 表的产生

由于数据通常以字节（当然也可以字的形式）形式出现，因此以 8 位数据产生所需要的 CRC 表，共计 256 个，以便在提高速度的同时可以节约存储器。产生表的过程就是分别求出从 0x00-0xFF 的 CRC 值，然后按照这个影射关系构成的一个数据表。当然，有两种办法来实现，一个是在运行时，通过调用 CRC 表程序来产生。其次，是用工具预先形成数据表，然后将其放在 ROM 中。

演示代码如下：

```
void GenCRC16Table()    //Calculation 256 CRC values without bit reflection
{
unsigned int i,j;
for(i=0;i<256;i++)
{
   CRC16Temp=i;
   CRC16Temp<<=8;
   for(j=0;j<8;j++)
   {
      if(CRC16Temp & 0x8000)
      {
        CRC16Temp<<=1;
        CRC16Temp ^=CRCPOLY16;
      }
      else    CRC16Temp<<=1;
   }
   CRC16Table[i]=CRC16Temp;
 }
}
```

## 3 总结

根据以上分析，无论在接收端还是发送端，CRC 的产生和校验都可以有一个 CRC 生成程序完成，这给程序带来了方便实现，简易性和快速性。根据实际的需要选用不同的方法。同样，32 位 CRC 的算法与 16 位大同小异，读者在理解 16 位 CRC 的基础上可以很方便的自行推导。

笔者

2005-7-19